Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

9. File Systems and I/O Management

9.1 Files and File Systems

- A **file** is a way to permanently store user and kernel code and data, typically on non-volatile storage like a hard disk. Secondary storage disks or hard disks are block devices that store contents of files across several blocks. In addition to the actual content of the file on disk, a file has several **metadata** associated with it: a name, location in a directory structure, location (i.e., addresses of blocks) on disk, a file type, access/read/write permissions, attributes like time of modification, file size, which users or processes have the file currently open, and so on. Users perform operations like creating, reading, writing/updating, deleting, renaming, truncating, and appending to a file. The reading and writing from a file can be sequential (from where the last read/write left off), or can be random access from a certain specified offset by seeking into the file at that offset. Most programming languages provide library functions to implement all of the above operations on a file. These library functions eventually call the file-related system calls provided by the OS.

- Files are typically organized into **directories**. Most operating systems treat directories as special files, where the content is simply a list of files and their metadata. The graph of directories and files can have one or more levels of depth; most modern operating systems permit upto a maximum configurable depth of directories, and a maximum limit on the number of files in a directory. At a very basic level, the graph of directories/sub-directories and files can be a simple tree. It can also have cross links if such linking functionality is supported by the OS.

- Every file/directory has data and metadata, stored across disk and memory caches. The data of a file can span several blocks on disk. The metadata of a file is stored in a data structure called the **inode**. The inodes in a file system have unique numbers, and an unnamed file can be uniquely identified by its inode number. When a file is not in active use, its metadata (inode) is stored on disk along with its content. When a file is actively being used, an in-memory copy of the metadata must be maintained. A **file control block or FCB or in-core inode** refers to an in-memory kernel data structure that stores a copy of a file's on-disk inode (along with some extra information) when the file is actively being read from or written to by any process. An inode cache in memory holds several in-memory inodes.

- A **directory entry** links a file name in a directory to its inode. That is, directory entries store mappings between file name and inode number for all files in the directory. Directory entries can be organized in disk blocks in several ways, depending on the OS implementation. Some operating systems provide the option of adding a link to one file/inode from multiple directories, creating cross-links between various branches of the directory tree. In such cases, multiple directory entries can exist for the same file in many directories. Broadly, operating systems provide two flavors of linking. With **hard linking**, a directory entry in the new location is

created to map a new file name and an existing inode number. With a less rigorous **soft linking**, a new directory entry is created to only map a new file name to the old file name, and the old file name must be looked up (if it exists) to locate the inode number.

- A file is only linked to its parent directory by default, during creation. However, multiple directories can subsequently create directory entries to (hard) link to this file, and can reference it with different names. The inode of a file maintains this link count. Deleting or unlinking (to be more precise) a file from a directory simply amounts to deleting the directory entry that points to the inode, and need not always result in the file being scrubbed from the disk. When multiple hard links exist to a file/inode from multiple directories, unlinking a file from one of the directories will still keep it accessible via the other links. A file is garbage collected only when it is unlinked from all its parents and its link count reaches zero. Note that this link count does not account for soft links: a file can be deleted even if some soft links exist to it; the soft links will simply become invalid at that time.

- The OS maintains system-wide and per-process **open file tables** for every file in active use. These open file tables store pointers to the in-memory inodes of the open files. When a user opens a file, a new entry is created in the **global system-wide open file table** to point to the in-memory inode of the file. If the file was already opened and an in-memory inode already exists, the file table entry simply points to the existing inode. If no in-memory inode is found, a new free in-memory inode is allocated to this file from the inode cache, and the in-memory version is populated from the disk version. Therefore, every open system call always creates a new global file table entry, but doesn't always allocate a new in-memory inode.

- The in-memory inode maintains a reference count to indicate the number of open file table entries pointing to it. (The reference count can also account for other things like current working directory pointers, as we will see later.) An in-memory inode is kept in memory as long as its reference count is non-zero, and can be recycled once all references to it have been closed. The in-memory inode cache typically reclaims/recycles such inodes with zero reference count when it runs out of space to store a new in-memory inode. Do not confuse the reference count of an in-memory inode with the link count of an inode in the directory tree.

- An open system call also creates an entry in the **per-process open file table**. The per-process table entry points to the system-wide file table entry, and can have a few other fields that are specific to the process itself. The per-process file table is usually maintained as an array of files in use by the process. When a user opens a file, the index of the newly created per-process file table entry is returned to the user. This index is called a **file descriptor or file handle**, and can be used by the process to refer to the file in all subsequent read/write requests on the file. The file descriptor number is local to a process because it is an index into the per-process (and not the system-wide) file table. When a new file descriptor must be allocated, most operating systems look through the array linearly to find the first unused index.

- System calls to open pipes and network sockets also create similar entries in the open file tables. However, these entries do not point to in-memory inodes. The file table entry for a network socket points to the socket data structures, and is used to send and receive network data via

sockets. Similarly, the file table entry for a pipe points to a pipe data structure, which contains some kernel memory to act as the pipe buffer. In fact, the pipe system call creates two file table entries in the global file table, and two file descriptors pointing to them. One file descriptor can be used to write into the pipe buffer, and the other can be used to read the written data, much like in a producer-consumer situation.

- Some file system functionalities require duplicating a file descriptor. When a file descriptor is duplicated, a second file descriptor is created to point to the same global file table entry. Therefore, the global file table entry has a reference count to keep track of the number of file descriptors pointing to it, and is garbage collected only when this count goes to zero. One example of this duplication is during a fork. When a parent process executes the fork system call, the parent initializes the child's file descriptors to point to the same global open file table entries using this mechanism of duplication. This allows both parent and child to access the same set of files. When a file descriptor is closed in one of the parent or child, the other can still access the file via its file descriptor, because the global file table entry is still valid. Some operating systems also provide a `dup` system call, which takes a file descriptor is a process as an input, and creates another file descriptor to point to the same open file table entry. Shells use this dup system call for various file descriptor manipulations.

- A **file system** is a way to organize files on secondary storage. A secondary storage disk typically has several partitions. A file system resides on a logical **volume**, where a volume can comprise of one or more partitions on a disk, or a complete disk, or even a collection of several disks. Every file system on a volume has a **super block or master file table** that stores a master plan for how the entire volume is organized, e.g., which blocks correspond to file data, which contain metadata, which blocks are free, and so on. There are several types of file systems, and all of them mainly differ in how they organize and store their data on disk, leading to different properties.

- A file system on a volume can store zero or more operating systems, as well as other user files. If the logical volume has an operating system, then a special block in the logical volume, called the **boot block**, will contain a boot loader that the BIOS can find. The boot block is typically block number 0, and is followed by the super block at block number 1. Note that the bootloader is stored in the boot block in a very simple fashion, without involving all the complexities of the full file system. This way, the BIOS can easily parse the bootloader. The boot loader, however, is more sophisticated, and can understand a file system's organization structure well enough to locate and load the kernel executable (or as is the case today, a more complex bootloader like GRUB) from disk into memory.

- Before a file system can be used to store and retrieve data, it must be *mounted*. Mounting a file system is equivalent to opening a file before reading/writing from it. Consider a file system of a certain type present on a volume on a disk. To start using this file system, one must provide an empty directory in the directory structure of the running system, and instruct the OS to mount the file system at this **mount point** in the directory tree. Once the file system is mounted and connected to the directory tree at a certain point, processes can access and work with the files on that file system. The OS maintains an in-memory **mount table** that lists all the file systems

currently mounted. Note that the first file system that holds the kernel executable is mounted at the root of the directory tree during boot time itself, so that the kernel executable can be read off the root file system during the boot process by the boot loader.

- A **disk buffer cache** is used to cache disk blocks in memory, and caches disk blocks corresponding to both file data and metadata. A buffer cache improves performance by reducing the number of accesses to a relatively slower secondary storage. In operating systems that use a disk buffer cache, the file system issues requests to read and write blocks to the buffer cache, and not directly to the hardware. If the request cannot be satisfied within the cache, a request to fetch the block from the hard disk is issued by the **device driver** to the **disk controller**. Note that some hard disk controllers may also implement some caching of disk blocks within the hardware itself; we only refer to the disk buffer cache maintained by the OS in our discussions. The device driver is the kernel's interface with the device hardware. Device drivers implement logic to start I/O operations on devices, as well as handle interrupts from the I/O devices, e.g., when the requested I/O operation completes.

- Finally, note that not all accesses to disks or files need to pass through the entire file system. Sometimes, the kernel may want to access raw blocks on the disk directly, without putting a file system's organizational structure on the disk. The swap space uses such **raw disk** access for speed and efficiency. Further, some applications like databases may want to perform **direct I/O** and bypass some (not all) of the functionalities of the file system like disk buffer cache, so that they can cache more intelligently within the application itself.

9.2 Layers of Abstraction in File Systems

- Systems calls are the interface provided to users to access files via file systems. Modern operating systems support several file systems (e.g., ext2, ext3, ext4, ZFS, and so on), each having a different set of properties, providing users with choices on which file system to use for what type of data. Since each file system stores and accesses its files differently, the implementations of system calls can be very different across file systems. To avoid this complexity, some operating systems like Linux provide a **virtual file system (VFS)** abstraction. VFS is a way to abstract out the implementation details of file systems, and arrive at a common skeleton for implementing a system call across file systems. VFS defines a set of abstract objects—files, directories, inodes, and superblocks—and operations that can be performed on each of these objects. Every file system that can be accessed via VFS must have data structures equivalent to the abstract objects, must implement the functions defined by VFS on the objects, and must register a pointer to these functions with the VFS subsystem. When a system call is made, VFS translates the system call into a set of function calls on the abstract objects, and invokes the registered file system functions accordingly.

- Note that the VFS abstraction integrates not just regular disk-based file systems, but also pipes, sockets, special file systems like `procfs`, and network file systems like NFS. Further, one can also implement special user space file systems, where the entity implementing the VFS functions is not kernel code but a user space process.

- To see a concrete example, let's examine how the open system call implementation would proceed in a kernel with a virtual file system. When the filename is received via the open system call, the VFS kernel code starts walking down the path of the filename. At each stage of the walk, VFS performs a lookup on a directory object, to locate the inode of the next directory or file in the path name. Note that, while the implementation of the lookup function is specific to each file system (e.g., it depends on how file entries are stored in the directory object), VFS is not concerned with these complexities, and simply invokes the lookup function from the pointer given to it. If the file to be opened is not found, a new file is created, an inode is allocated for it, and a pointer to it is inserted into the directory object. Once an inode corresponding to a filename is identified (or a new one created), VFS installs a pointer to it in the open file table, and returns the file descriptor. Similarly, other system calls to read and write data from a file will cause VFS to invoke the corresponding functions on the inode object of the file, which are implemented by the underlying file system.

- Note the *layering* and various levels of *abstraction* that go into the design of modern file systems. The OS exposes the system call interface to user programs and libraries. The system call interface is implemented by a virtual file system, which defines an interface consisting of a set of generic objects and functions, and translates the system calls into function calls on these generic objects. Actual file systems implement the functions called by VFS, and issue calls to read/write blocks from the hard disk. These calls then pass through a buffer cache, which first tries to serve the disk read/write requests from the cache memory itself, and passes them over to the actual disk hardware only in the case of a cache miss. When blocks have to be fetched from the disk, the device driver interfaces with a disk controller to fetch the content of specific block

numbers from the disk hardware into the main memory. Each layer provides an abstraction to the layer above it, and hides the implementation details of the lower layers. Each layer also depends on a service provided by the lower layer to simplify its job. A file system is a classic case study of how a modular layered design can greatly simplify what would otherwise have been a very complex OS subsystem.

9.3 Design Choices for File and Directory Metadata

- A block is a typical granularity of allocation of storage space on a hard disk. A file may span several such blocks. The most important piece of metadata that must be tracked for a file is its location on disk, i.e., which blocks on disk hold the file's contents. How does the file system allocate and track the various blocks that belong to a file? There are several ways of allocating blocks to a file, and this problem is similar to that of allocating memory to a process. One way is **contiguous allocation**, where a free set of blocks big enough to accommodate the entire file is found, and the file is stored contiguously in those blocks. This type of allocation is simple, and the file system has to maintain very little information (starting block number, and size) in the inode or any other metadata structure to locate a particular file. However, this allocation suffers from external fragmentation. Further, it creates problems when a file size changes drastically from its initial allocation. Older file systems that used this technique often had to be taken offline and defragmented to create contiguous blocks over time.

- Another technique for allocating and keeping track of data blocks in a file is **linked allocation**, where blocks are allocated in a possibly non-contiguous manner, but a pointer is maintained from a block to its next. This pointer to (i.e., block number of) the next/previous blocks can be maintained on the disk block itself, but accessing a file at a certain offset may cause several disk reads to even locate the correct block number. An alternative technique is to maintain a **file allocation table (FAT)**. A FAT has one entry for every block (or a cluster of blocks) on the disk. Each entry indicates if the block is free or if ti has been allocated to some file. If the block has been allocated, the FAT entry contains the number of the next block of the file (or indicates that this is the last block). With a FAT in place, one only needs to store the starting block of the file in the file metadata, and simply follow pointers in the FAT to locate all the other blocks. The FAT is usually kept in memory. The number of entries in the FAT is roughly equal to the number of usable blocks on disk, and the number of bits to store a block number in each FAT is proportional to the logarithm of the number of blocks. The maximum size of the FAT table entry limits the size of disks that can be managed with FAT. While newer methods have replaced linked allocation with FAT, this design is still robust enough to be useful in simple file systems, especially when only sequential file access is desired.

- Another technique to allocate and track data blocks is called **indexed allocation**. With this allocation method, all the pointers to the data blocks of a file are stored in a data structure called the **index node** or **i-node**. This technique is so prevalent in Unix-like operating systems that the term "inode" has become synonymous with the data structure containing the metadata of a file itself. The index node holds the block numbers of all the blocks of a file, i.e., the i-th entry holds the number of the i-th data block. So, for each file, the file systems needs to only store the block number where the index node is stored, using which it can locate all the other data blocks of the file. But what if the file is so large that all the block numbers do not fit in one index block? One could store the indices across many disk blocks, and store pointers from one index block to the other to form a linked list of index blocks. Or, one could build a multi-level index, where the top-level index block stores the block numbers of the next-level index blocks, and the last tier of index blocks store the actual block numbers of the file data. The index node structure

of Linux is the variant of the above idea. The index block in Linux stores the block numbers to the first few blocks of the file (also called **direct blocks**). If the file size is larger than what can be stored in the direct blocks, the index block contains the block number of a **single indirect block**, which in turn stores the block numbers of the next few blocks of the file. If the file size is even larger, the index block contains pointers to a **double indirect block** and a **triple indirect block** as well (should it be required). For small files, reading the index block from disk should be enough to find the addresses of the first few direct blocks of the file. For larger files, multiple index blocks have to be retrieved before the actual data block can be located.

- Another design problem in file systems is that of keeping track of the free unallocated blocks on a disk. File systems that use FAT automatically handle free space management as well using FAT, while other file systems use a variety of techniques. One technique is to use a **free block bitmap**, where one bit per block indicates if the block is free or not. These bitmaps can, however, get very large for large disks, especially if one has to store them in memory. Another way is to keep a **linked list of free blocks**, where each free block stores the number of the next free block. The OS must maintain pointers to the first (to allocate a new block) and last (to add released blocks) free blocks handy. A variant of this method is to store the addresses of free blocks in special disk blocks, and maintain a linked list of these clusters of free block pointers. Yet another technique is to store information about a contiguous set of free blocks by storing the starting block number and count. These address/count pairs can be stored in a linked list, or in a more efficient structure like a balanced tree.

- Let us now examine the design choices in implementing directories. At the very basic level, a directory can simply be a file that stores a **linear list** of the file names and a pointer to the corresponding inodes. However, this design is not optimal because most file operations require traversing the entire list. For example, an operation to lookup a specific file would require traversing the entire list in the worst case. (To mitigate this problem, some file systems augment the linear list with a hash table, where the hash table stores a mapping between the filename and its position in the linear list.) Alternatively, a **hash table** (with provision for collisions) can directly store a mapping from the filename to its inode number, and a directory simply contains a representation of this hash table on disk or in memory. Another way to enable quick lookups of files in directories is to store the filenames in a sorted list (sorted by name), using **balanced binary search trees**. Directory implementations based are balanced binary search trees (like B trees and B+ trees) are some of the most common ones found in modern operating systems.

- File system designs place constraints on maximum file sizes and disk sizes that can be managed. For file systems that allow random access, data in a file can be read or written by providing a byte offset into the file. In such cases, the maximum size of the file is constrained by the number of bytes that can be addresses by such pointers. For example, 32-bit offsets can address a maximum file size of 4GB. Other factors like the number of bits available to store block numbers in inodes effect the maximum size of disks that can be managed by a file system. For example, if a file system has a FAT with 12-bit entries, and each entry pointed to an 8KB cluster, then the FAT would only allow access to a 32MB hard disk. Similarly, a file size in Linux is also constrained by the number of data blocks that can be stored in the inode as direct blocks, and in the single/double/triple indirect blocks.

9.4 Filesystem Consistency

- Consistency is a big concern when manipulating metadata on the file system. While correct file system implementations never wilfully leave the file system in an inconsistent state, system crashes could lead to inconsistencies in file system data. To see why, note that most file system operations update several blocks on disk, including several blocks of metadata. For example, when creating a file, a free block is marked as allocated, the file's inode is allocated and updated to point to the new data block, and the directory containing the file is updated to store a pointer to the new inode. If the system crashes before all of these updates complete, the file system can be left in an inconsistent state. For example, a free block could have been marked as allocated, but neither the inode nor the directory could have any pointer to it. Worse still, the inode could be pointing to a data block, but the system could have crashed before the data block was marked allocated, leading to the block potentially being allocated again. These consistency issues are further compounded by the fact that some metadata blocks reside in memory and some on disk, and not all metadata changes in memory would have been flushed to disk at the time of a system crash.

- One simple way to fix consistency issues is to check the file system for inconsistencies when booting after a crash. File system utilities like `fsck` check all file system data structures for dangling pointers and such. Further, file systems also make efforts to avoid inconsistencies in the face of crashes in several ways. File systems always perform metadata updates in an order that is more resilient to crashes. For example, a free block is marked as allocated before adding a pointer to it from an inode. File systems also flush disk writes corresponding to metadata to the disk quickly, instead of letting them stay in the memory cache for long. Some file systems periodically take a snapshot of file system metadata, and never overwrite important metadata, so that the system can be restored to an older consistent state after a crash.

- A popular technique for ensuring consistent file system updates is to use a log-based or transaction-oriented or journaling file system. In a log-based file system, various metadata updates that correspond to one file system operation (such as file creation) are grouped into a **transaction**. A transaction is a set of updates that must be performed *atomically*. For example, the steps of allocating a free data block, pointing to it from the inode, and pointing to the inode from the directory could constitute a transaction that creates a new file. Log-based file systems do not make metadata updates directly on to the disk. Instead, each transaction is written to a log file on the disk. Once all metadata changes in a transaction are written to the log, a commit is written to the log. Now, the file system installs all the block changes in the transaction on to the actual data and metadata blocks on disk. Once all the changes in a transaction have been installed, the transaction is deleted from the log. When the system recovers from a crash, it discards any uncommitted log entries (because they would have recorded incomplete changes) and replays any committed transactions in the log that haven't been installed yet. Thus, all the updates in a transaction are fully present in the file system (when a committed transaction is installed/replayed after crash), or they are altogether absent from the file system (if the system crashed midway before a transaction committed), leading to the atomic property of transactions. Note that log-structured file systems provide strong consistency guarantees but impose a performance overhead due to all the extra disk operations required in maintaining the log.

- Given the various choices available in each aspect of file system design, it is not surprising that several file systems exist today. Modern operating systems support multiple file systems simultaneously using the virtual file system interface. Therefore, the choice of which file system to use depends on the user's expectations of performance and reliability.

9.5 Buffer Cache and Memory-mapped Files

- Any disk access will involve some caching in main memory, leading to interactions between the file system and the memory subsystem via caches. The nature of the interaction depends on how a file is accessed. In general, there are two ways to read and write data from a file: using `read` and `write` system calls, or accessing a file by memory-mapping it using the `mmap` system call. Let us consider the implications of each of these techniques below.

- The normal way to access a file in a user program is to open it and get a file descriptor, then use the file descriptor to read and write data at a certain offset in the file. The file system maps this offset to a block number on disk using information in the file inode, and fetches the corresponding disk blocks into kernel memory. If the operation is a read, the disk block content that is fetched into the kernel is once again copied into user space memory provided by the user application. If the operation is a write, the copy of the disk block fetched from disk is updated in memory, and subsequently flushed to the disk. Now, given that a copy of the disk block is already in memory, it makes sense to save it for a little while longer, in order to avoid going to disk for the same block again in the near future. Therefore, the **disk buffer cache** is used to cache disk blocks in main memory. For each disk block that was recently accessed, the disk buffer cache stores the device identifier, block number on the device, and contents of the disk block. Read requests are served from cache when possible, and a block is fetched from disk only if a valid copy is not found in the cache. On a write system call, the buffer cache updates the contents of the block in the cache, and marks it as **dirty**. Dirty blocks are then flushed to the disk via the device driver.

- Read system calls for disk blocks are blocking if the block is not found in the buffer cache and must be fetched from disk. However, reads for blocks found in the cache can return immediately. Write system calls can be blocking or non-blocking. If the application write system call simply writes to the buffer cache and returns to the user program, it is called an **asynchronous write**. Alternately, if an application chooses to block on a write till the write reaches the disk, it is called a **synchronous write**. Operating systems usually provide both flavors of the write system call. While asynchronous writes can be used by applications desiring good performance, synchronous writes can be used for important disk data, e.g., file metadata blocks, where losing the written data that is in memory due to a system crash can be catastrophic. A disk buffer cache that flushes dirty blocks to the disk asynchronously, without causing the writing process to block, is called a **write back cache**. Alternately, a buffer cache that causes the writing process to block till the time the write is saved on disk is called a **write through cache**.

- How much memory can the buffer cache use? There is an upper limit on how many disk blocks can be in cache, because main memory is much smaller than disk. When the buffer cache runs out of space, it reuses the least recently used (LRU) disk blocks by evicting the old block and replacing it with the contents of a new disk block. If the block being evicted is dirty, it must be flushed before being reused. In modern operating systems, disk buffer caches can grow very large to occupy most of the free physical memory that is not in use by other processes: all memory that is not used by the kernel or user processes is put to good use by the buffer cache.

- Using a disk buffer cache has several benefits. The performance of disk-bound applications can significantly improve, especially if the hit ratio in the cache is high. Multiple processes that access the same disk blocks can avoid going to the disk every time. In fact, popular disk blocks like file system super blocks and important file metadata are almost always found in the cache, leading to good performance on file system operations. The buffer cache also synchronizes disk access across multiple processes, by ensuring that there is only one copy of the disk block in the entire system that is visible across all processes. Using synchronization mechanisms, it also ensures that only one process can edit a certain block at a given time. For example, if two processes simultaneously attempt to write to the same block on disk, both writes will access the same disk block buffer in memory, and the writes will be serialized in some order by the cache, instead of overwriting each other.

- Another way to access files is by using the `mmap` system call. When a file is memory-mapped, the kernel makes the file a part of the virtual address space of the process by storing the file contents into physical memory pages, and by updating the page table entries of the process to point to these new pages. Thus, when a file is memory-mapped, file contents are fetched from disk in page-sized chunks. Now, reading from and writing to the file is simply equivalent to accessing and updating memory variables in the program.

- When reading a memory-mapped file, the file contents are first fetched into memory pages from the disk. The user-space process then directly accesses the kernel pages that store the file contents, avoiding an extra copy of data from kernel space to user space. This is one of the reasons why memory-mapped reads can be slightly faster than regular reads for large files. However, if a user program wishes to read only a small amount of data, memory-mapping may not lead to any tangible performance benefits.

- When writing to a memory-mapped file, the changes are made to the memory page holding the file contents to begin with. Subsequent actions depend on the mode in which the file is memory mapped. If the file is memory mapped in a "private" mode (i.e., changes visible only to the process), the writes are stored in memory, and flushed to disk only upon unmapping a file. Thus, applications that write to the disk several times will benefit from memory mapping a file, because writes can be very fast. The benefits of memory mapping are less apparent when the file is opened in a shared mode, where changes are flushed to disk more often in order to be visible to other processes.

- Now, when a page-sized chunk of a file is read for a memory-mapped read, the request first goes to the buffer cache. Thus, the blocks that make up the page exist in the buffer cache portion of the memory, and also in the "page cache" (which is simply the set of memory pages available for allocation to user processes). Having two copies of the same content in memory due to this double caching is wasteful. Therefore, modern operating systems have a **single unified page cache** instead of separate caches for memory pages and disk blocks. The unified cache consists of memory pages that contain disk contents, as well as other pages that are not backed by disk. Disk blocks are directly read into page-sized areas of memory, and the disk buffer cache only holds pointers that map a specific disk block number to the location of the block's content in the page cache.

9.6 Kernel I/O subsystem

- We now look at how the kernel communicates with the hardware of a secondary storage disk to give out read/write instructions and transfer data. While we have looked at only hard disks so far in the discussion of file systems, the general principles discussed here apply to all I/O devices. That is, we will be studying the design of the **kernel I/O subsystem** as a whole. This subsystem consists of **device drivers**, and device-independent code. The device drivers understand the device hardware internals and communicate with the device in a language that it follows. For example, a disk device driver would know how to instruct the disk to fetch a certain block. The device independent part of the kernel code handles generic I/O functions, e.g., higher layers of the file system stack or the networking stack.

- I/O devices are of two main types: **block devices** (e.g., hard disks), and **character devices** (e.g., keyboard, network interfaces). With block devices, data is stored and addressed on the device at the granularity of blocks. That is, the kernel can request the device to store or retrieve data at a specific block number. Character devices simply accept or deliver a stream of characters or bytes to the kernel; there is no notion of a permanent block of data that can be addressed. In other words, you can perform a *seek* on a block device to point to a specific block number to read from or write to, while with character devices, you simply read or write at the current location in the stream.

- Irrespective of the type of device the kernel is communicating with, the user programs see a uniform interface consisting of the `read` and `write` system calls on a file descriptor to communicate with I/O devices. The system calls to *initiate* communication with the device, however, may differ slightly. For example, one has to open a file to communicate with the disk, while one has to use the `socket` system call to open a network connection. In addition, the `seek` system call can also be used to reposition the offset to read from or write to for block devices. Finally, the `ioctl` system call can be used to pass any other instruction to the device that is not a simple read/write/seek/open.

- Communication with an I/O device can be blocking or non-blocking, depending on the options set for the particular I/O communication. When a process chooses to block on a read or write system call, until the I/O operation completes, such an I/O is referred to as **synchronous I/O**. In contrast, **asynchronous I/O** can refer to a process using non-blocking read/write operations (possibly with polling to check for status of completion), or special system calls or libraries to enable optimized event-driven communication with I/O devices. For disk operations, a blocking read makes the most sense, since the process cannot make much progress without receiving the disk data. The write to the disk, however, can be blocking on non-blocking, depending on whether the application needs to make sure that the write has reached the disk or not. Typically, applications use blocking writes for sensitive, important data to the disk, and non-blocking writes for regular data.

- A note about networking-related system calls. The `socket` system call creates a communication socket, using which applications can send and receive data. Every socket has associated transmit and receive buffers which store application data. The `read` operation on a network

socket is blocking by default if there is no data in the receive buffer. A process that requests a read will block until data arrives from a remote host. When the data arrives via an interrupt, the TCP/IP network stack processing is performed on the packet as part of the interrupt handling, and the application data is placed into the receive buffer of a socket. The process blocked on the read is then woken up. A non-blocking option can be used on a socket to not block on reads, in case the applications wants to do other things before data arrives. In such cases, the application must periodically check for data on the socket, either by polling, or by using event-driven system calls like `select`.

- When a process calls `write` on a network socket, the data from the transmit buffer undergoes TCP/IP processing and is handed over to the appropriate device drivers. By default, network writes do not block until the data is transmitted, and only block until data is written into the send buffer of the socket. A transport layer protocol like TCP in the kernel is responsible for in-order reliable delivery of the data, and applications do not need to block to verify that the written data has reached the remote host. Writes can, however, block if there is insufficient space in the socket's transmit buffer due to previous data not being cleared (e.g., due to TCP's congestion control). Writes on non-blocking sockets will fail (not block) if there is insufficient space in the socket buffer, and the application must try later to write. Additional system calls `connect`, `accept`, `bind`, and `listen` are used to perform TCP handshake and connection setup on a socket, and must be performed prior to sending or receiving data on a socket.

- Every devices connects to the computer system at a logical **port** and transfers data to and from the host system via a **bus** (e.g., PCI bus, SATA, SCSI). Ports and buses have rigidly specified protocols on how data is marshalled. Most I/O devices have a special hardware called the **device controller** to control the port and bus, and manage the communication with the main computer system. Device controllers typically have two parts, one on the host and the other on the device. The host controller or the host adapter plugs into the motherboard's system-wide bus (that connects the CPU, memory, and other device controllers), and provides the port abstraction by transferring instructions and data from the CPU onto the device bus. The part of the controller that resides on the I/O device is responsible for transferring instructions/data between the bus and the actual device hardware. Some device controllers also implement intelligent optimizations on top of the simpler hardware.

9.7 Communication and Data Transfer with I/O Devices

- How does the kernel (or rather, the device driver) communicate instructions to I/O devices? All communication to the device must go through the port on the host controller. Conceptually, each port has 4 types of registers (actual details may differ across devices):
  - A **data-out** register for data from the host to the device.
  - A **data-in** register for data from the device to the host.
  - A **control** register written by the host and read by the device
  - A **status** register written by the device and read by the host.

  For example, to start a disk read, the driver puts the block number in the data-out register, and sets the control register to start the read. When the disk block is ready, the device sets the status register and places in the block content in the data-in register.

- Now, how are all these I/O registers in a port accessed? There are two ways:
  - There could be special hardware instructions to read and write specific contents into these registers. This is called **isolated I/O**. For example, the `outb` and `inb` (or equivalent) instructions in x86 are used to transfer data from CPU registers to specific output ports.
  - Modern operating systems use **memory-mapped I/O**, where the registers of various devices are mapped into the kernel address space at specific addresses. The CPU can then use regular instructions that read from or write to memory to access I/O registers as well. Memory-mapped I/O is generally faster because the CPU can write larger chunks of data with this method than it can through special instructions. Accesses to parts of the address space reserved for I/O devices are detected at the MMU, and are served from the device instead of from the RAM.

- Once the instructions have been communicated using one of the two techniques above, how does the CPU monitor the progress of the execution of the instruction? Conceptually, there are two ways:
  - The CPU could busily wait for the device to get back with a status, by constantly checking the status register. This is called **polling**. For example, when the CPU instructs a hard disk to fetch a particular block, it could monitor the status register continuously. When the status indicates that the read is complete, the CPU can proceed to read the disk block from the data-in register. Note that this technique is highly inefficient, as I/O devices are much slower than CPUs, resulting in a large number of cycles wasted in busy-waiting. The bootloader is typically the only place in modern operating systems where polling is employed, simply because the CPU has nothing better to do until the bootloader loads the OS.
  - Alternately, the CPU could instruct the hardware to raise an interrupt when the I/O instruction is completed. This is called **interrupt-driven I/O**. An advanced programmable interrupt controller (APIC) is a specialized piece of hardware that monitors the status of

I/O devices, and multiplexes interrupts from all I/O devices into one of the several interrupt lines available. The APIC is usually split into two components, an IOAPIC that resides on the I/O device, and a local APIC (LAPIC) that resides on each CPU core. The APIC can be programmed to direct interrupts from certain I/O devices to certain CPU cores. The LAPIC also implements a high resolution timer, so that each CPU core independently gets its timer interrupts. By using a specialized piece of hardware to monitor the status of I/O devices, the CPU frees up its cycles to execute other tasks instead of waiting for I/O to complete.

When an interrupt occurs, a CPU instruction (e.g., `int` in x86) looks up the interrupt descriptor table (IDT), and invokes the kernel's interrupt service routine/interrupt handler for that particular interrupt. Interrupt handlers of I/O devices are typically implemented by device drivers that understand the specifics of the particular device, so the kernel's interrupt handler typically invokes a function in the corresponding device driver. For example, when the disk controller interrupts the CPU to indicate a completed disk read, the disk driver copies the disk block contents from the device registers to the kernel memory, and possibly initiates the next request to the disk (if more are waiting).

Recall that interrupt handling in Linux is split into a top-half that executes immediately, and a bottom half that is deferred and executed at a later time. For example, the top half typically transfers data from the hardware to kernel, and starts the device on its next task, in order to fully utilize the slow I/O device. Bottom halves do less performance-critical tasks like TCP/IP processing of the received packet, or waking up a process that is blocked for the particular disk block.

- The polling and interrupt-driven modes of managing I/O devices are primarily programmed into and executed by the way of CPU instructions. Hence these two modes are also referred to as **programmed I/O**. Even in the interrupt mode, programmed I/O still consumes significant number of CPU cycles to copy data from device memory to kernel memory. Moreover, the data transfer occupies precious space in CPU caches. To avoid burdening the CPU with this somewhat mundane task of copying data, modern operating systems rely on a technique called **direct memory access (DMA)**. DMA operations are handled by a specialized hardware called the DMA controller. A certain section of the kernel memory is designated for DMA-based I/O operations by the kernel. To start an I/O operation, say, a disk read, the CPU instructs the DMA controller to start the I/O operation, and gives it a pointer to a chunk of memory in the designated DMA zone. When the I/O operation is complete and the data is available on the device, the DMA controller requests access to the system bus to access main memory, and directly copies data from the device to the designated DMA zone in main memory. The CPU is not involved in the copying of data, and is free to execute other instructions, except for the restriction that the CPU cannot access the memory being used by the DMA controller while the memory transfer is in progress. Once the DMA controller finishes the memory transfer, it raises an interrupt to let the CPU know that the data from the device is available. Interrupt handling then proceeds as usual (e.g., start the next disk operation, awaken sleeping process, and so on), except for the fact that the interrupt handler directly gets a pointer to the received data in kernel memory, without having to copy it from the device itself. DMA is very useful in improving performance, especially for I/O devices like hard disks and network cards that read and write

significant amounts of data. Note that using DMA does not avoid interrupts altogether; instead, it enables interrupt handlers to finish faster by avoiding the data copy overhead.

- Note that interrupt handlers or DMA controllers transfer data between devices and kernel memory, not user memory. That is, when a process wishes to read a disk block, the data is first fetched into the kernel memory, and subsequently copied into the user provided buffer when read returns. Similarly, when a process writes to an I/O device, the content to be written is first copied into kernel memory from user memory, and communication with the I/O device is initiated thereafter.

- Applications that perform significant amounts of I/O (e.g., disk-bound applications or network applications) are typically bottlenecked by the speed at which I/O operations can complete. I/O operations impose two types of overheads. First, there are many transitions from user mode to kernel mode, for processing systems calls and interrupts. Also, I/O bound processes block frequently, causing several context switches. These transitions from user mode to kernel mode, and from the context of one process to that of the other, waste CPU cycles, leading to suboptimal performance. Second, I/O operations typically require copying data multiple times. For example, when a disk block or network packet is read, it is copied once from device memory to kernel memory (by the interrupt handler or the DMA controller), and once from kernel memory to a buffer in user space (when the process performs a read system call). These effects are more pronounced in virtualized systems, where one has to deal with these overheads twice, once in the guest OS and once in the host OS.

- Modern computer systems use several techniques to mitigate the impact of these issues and improve I/O performance. These techniques roughly fall under two umbrellas: reducing the performance impact of handling interrupts, and striving for zero-copy packet transfer (where unnecessary packet copies are avoided). For example, modifications have been proposed to the kernel I/O subsystem to batch the handling of interrupts and systems calls, thereby avoiding frequent switching between CPU modes. Some high-performance device drivers (e.g., NAPI or new API drivers) resort to polling at regular intervals, instead of using interrupts, to handle several I/O requests in one batch. Some network interface cards uniformly distribute interrupts across cores (while ensuring that packets of one flow go to one core only, to preserve cache locality), to avoid overloading a single core with all networking interrupts. Some modifications to the kernel (e.g., netmap framework) memory map a region of kernel space into user memory, to directly let users access/manipulate data in kernel buffers during the read/write system calls. Further, some kinds of specialized I/O hardware directly copy data from the device to memory mapped regions in user space. Improving the performance of the kernel I/O subsystem in the context of modern applications is a rapidly advancing research topic, and a detailed study of these techniques is beyond the scope of this course.

9.8 Secondary Storage Disks

- Hard disks today comprise of a set of magnetic platters, each with a read/write head that moves over the platter, to store bits of information. A platter consists of several concentric tracks, and each track has several sectors. The typical size of a sector is 512 bytes. A sector is also referred to as a block, and is the granularity at which data is stored and addressed by the OS. The disk controller exposes the disk as a set of **logical blocks** to the kernel, and maps logical block numbers to actual physical location on the disk internally, in order to hide the physical details of the disk from the operating system. Occasionally, the disk controller can also shift a logical block from one physical sector to another on the disk, without the operating system noticing it, when some sectors on the disk go bad. Note that disks are mechanical devices, with a read/write head that physically moves over a rotating platter, making hard disks somewhat prone to failures and data loss. The disk controller, the device driver, and the file system must all be able to gracefully handle such failures.

- When the disk controller receives a request to read/write a certain block, it must first reposition the read/write head to the actual location of the block. This involves moving the disk head to the appropriate track/cylinder, and then waiting for the disk to rotate so that the required sector comes under the read/write head. The time taken for the former operation is called the **disk seek time**, while the time taken for the latter is called the **rotational latency**. Typically, the disk seek time is the largest component in the time required to access a random disk block, and the actual time taken to transfer data sequentially from a set of contiguous blocks is quite small relative to the seek time to reposition the disk head. Therefore, disks are much faster when performing sequential I/O rather than when doing random access I/O.

- When a queue of requests builds up for disk I/O, a naive technique to serve them would be in a first-come-first-serve (FCFS) order in which they arrived. However, this way of scheduling requests can cause the disk head to spend more time seeking than actually transferring data. In order to optimize disk performance, I/O-bound workloads benefit from **disk scheduling algorithms**, which reorder pending requests in the disk queue to maximize disk performance. These scheduling algorithms can be implemented by the device driver or the disk controller or both. Some popular algorithms are described below.

  - The **SSTF (Shortest Seek Time First)** algorithm selects the next disk request as the request whose block is closest to the current position of the read/write head, thereby minimizing disk seek time. This is conceptually similar to the shortest job first (SJF) scheduling algorithm. This algorithm has the problem that it can lead to starvation of disk requests that are farthest from the current head position, as newer closer requests may get priority indefinitely.
  - In the **SCAN** algorithm, the disk head moves from one end of the disk to the other and back, servicing any requests pending along the way. This is also called the elevator algorithm, since the disk head behaves like an elevator. This algorithm has the disadvantage that when the disk reaches one end and starts servicing requests on its way back, the requests on that end of the disk see a much lower waiting time (since the disk just passed it

in one direction earlier), while requests on the other end of the disk see a much larger wait time. A modification to the SCAN algorithm is the C-SCAN algorithm, where the disk, on reaching one end, immediately turns back and starts servicing requests from the other end, without pausing to service requests on its return trip. In this way, the disk is simply treated as a circular list, where one goes from the end to the start in a single step. Other variants of the SCAN and C-SCAN algorithms are the LOOK and C-LOOK algorithms, where the disk head only goes as far as the last request in each direction, and not until the end of the disk.

- How are bits stored on the disk? Most disks are formatted during the manufacturing process (and even periodically by the user), so that each block has a header/trailer, with a simple check sum or error correcting code or parity bits written into it. While these extra bits can detect and even correct a small number of bit errors, some errors are not recoverable, and can lead to bad sectors. These failures must be handled at various levels, by the disk controller, as well as by the file systems. For example, disk controllers can allocate a logical block to another physical sector, to bypass bad sectors. Some file systems implement their own checksums and integrity checks to detect and guard against disk errors.

- Finally, many advanced storage solutions are available in the market, beyond simple storage disks. RAID (Redundant Array of Inexpensive Disks) is a disk organization where multiple disks are used to mask failures that result from storing data on any one disk. RAID storage systems have several levels of protection, where one can tradeoff storage overhead with resilience against failures. For example, one can add a large number of parity bits to be robust to a larger fraction of disk failures. NAS (Network Attached Storage) is another solution being used for remote and robust storage in enterprise systems.